A Datatype for Binary Trees

One of many possible definitions (no interior values)

```plaintext
datatype bi_tree =
    Leaf of int
  | Node of bi_tree * bi_tree;

Possible constructors:

- Leaf;
val it = fn : int -> bi_tree
- Node;
val it = fn : bi_tree * bi_tree -> bi_tree
- Leaf(4);
val it = Leaf 4 : bi_tree
- Node(Leaf(3), Leaf(4));
val it = Node (Leaf 3, Leaf 4) : bi_tree
```
Recursion Template

```
datatype bi_tree = 
  Leaf of int 
  | Node of bi_tree * bi_tree;
```

Recall our guiding principle:

```
A recursive function 
follows the structure 
of inductively-defined data.
```

Recursion template:

```
fun tree_rec(Leaf(n)) = ....  
  | tree_rec(Node(bt1,bt2)) = 
      ... tree_rec(bt1)...
      ... tree_rec(bt2)...;
```
Sum of Tree Data

Sample tree:

```plaintext
val t = Node(Node(Leaf(3), Leaf(4)), Leaf(1))
val t = Node(Node(Leaf #, Leaf #), Leaf 1)
```

Desired behavior

```plaintext
use "tree_library.sml"
...
val tree_sum = fn : bi_tree -> int
val tree_sum(t);
val it = 8 : int
```

Implementation:

```plaintext
fun tree_sum(Leaf(n)) = n
  | tree_sum(Node(bt1, bt2)) =
    tree_sum(bt1) + tree_sum(bt2);
```
Flipping a Tree

- val t = Node(Node(Leaf(3), Leaf(4)), Leaf(1));
- val t = Node(Node(Leaf #, Leaf #), Leaf 1) : bi_tree

Desired behavior

- use "tree_library.sml";
...
- val tree_flip = fn : bi_tree -> bi_tree
- tree_flip(t);
- val it = Node
  (Leaf 1, Node (Leaf #, Leaf #)) : bi_tree

Implementation:

fun tree_flip(Leaf(n)) = Leaf(n)
 | tree_flip(Node(bt1, bt2)) =
    Node(tree_flip(bt2),
         tree_flip(bt1));
Fringe of a Tree

```
val t = Node(Node(Leaf(3), Leaf(4)), Leaf(1));
val t = Node(Node(Leaf #, Leaf #), Leaf 1)
: bi_tree
```

**Desired behavior**

```
use "tree_library.sml";
...
val tree_fringe = fn : bi_tree -> int list
- tree_fringe(t);
val it = [3,4,1] : int list
```

**Implementation:**

```
fun tree_fringe(Leaf(n)) = [n]
| tree_fringe(Node(bt1, bt2)) =
    tree_fringe(bt1)
  @ tree_fringe(bt2);
```
A Datatype for General Trees

Trees with arbitrary branching (still no interior values)

```plaintext
datatype 'a gtree =
  Leaf of 'a
  | Node of 'a branches

and 'a branches =
  Empty
  | Branch of 'a gtree \* 'a branches;
```

A recursive function still follows the structure of inductively-defined data.

Recursion template now involves mutual recursion:

```plaintext
fun gtree_x (Leaf(n)) = ...
| gtree_x (Node(bs)) = ... branches_x(bs)...

and branches_x (Empty) = ...
| branches_x (Branch(gt,bs)) =
  ... gtree_x(gt).... branches_x(bs)....;
```
Sum of Tree Data

datatype 'a gtree =
    Leaf of 'a
  | Node of 'a branches
and 'a branches =
    Empty
  | Branch of 'a gtree * 'a branches;

fun gtree_sum (Leaf(n)) = n
  | gtree_sum (Node(bs)) = branches_sum(bs)
and branches_sum (Empty) = 0
  | branches_sum (Branch(gt, bs)) =
      gtree_sum(gt)
      + branches_sum(bs);
Lifting to Higher-Order Functions

```ocaml
datatype 'a gtree =
    Leaf of 'a
  | Node of 'a branches
and 'a branches =
    Empty
  | Branch of 'a gtree * 'a branches;
```

As for lists, we can write a `map` function

```ocaml
fun gtree_map f (Leaf(n)) = Leaf(f(n))
  | gtree_map f (Node(bs)) =
      Node(branches_map f bs)
and branches_map f (Empty) = Empty
  | branches_map f (Branch(gt, bs)) =
      Branch(gtree_map f gt,
             branches_map f bs);
```

```ocaml
val gtree_add1 = gtree_map (fn x => x + 1);
```

What about `filter` or `fold`?